

Combining Fixed-Point Definitions and Game Semantics in Logic Programming

Keehang Kwon

Dept. of Computer Engineering, DongA University
Busan 604-714, Korea
khkwon@dau.ac.kr

Abstract: Logic programming with fixed-point definitions is a useful extension of traditional logic programming. Fixed-point definitions can capture simple model checking problems and closed-world assumptions. Its operational semantics is typically based on intuitionistic provability.

We extend the operational semantics of these languages with game semantics. This extended semantics has several interesting aspects: in particular, it gives a logical status to the *read* predicate.

keywords: interaction, game semantics, read, computability logic.

1 Introduction

Logic programming with fixed-point definitions is a useful extension to the logic of Horn clauses. In this approach (see, for example, [6, 10]), clauses of the form $A \triangleq B$ – called *definition* clauses – are used to provide least fixed-point definitions of atoms. We assume that a set \mathcal{D} of such definition clauses – which we call program – has been fixed. The following *definition-right* rule, which is a variant of the one used in LINC[10], is used in this paper as an inference rule which introduces atomic formulas on the right.

$$pv(\sigma, \mathcal{G} \vdash A) \text{ if } A' \triangleq B \in \mathcal{D} \text{ and } A'\theta = A\sigma \text{ and } pv(\sigma\theta, \mathcal{G} \vdash B).$$

This rule is similar to backchaining in Prolog with the difference that an answer substitution σ is maintained and applied to formulas as *lazily* as possible here. The *definition-left* rule is a case analysis in reasoning.

$$pv(\sigma, A : \mathcal{G} \vdash D) \text{ if, for each } \theta \text{ which is the } mgu(A\sigma, A') \text{ for some } A' \triangleq B \in \mathcal{D}, pv(\sigma\theta, B : \mathcal{G} \vdash D).$$

This rule is well-known and used to instantiate the free variables of the sequent by θ , which is a most general unifier (mgu) for atoms $A\sigma$ and A' . If there is no such θ , the sequent is proved.

The operational semantics of these languages is typically based on intuitionistic provability. In the operational semantics based on provability, solving the universally quantified goal $\forall xD$ from a definition \mathcal{D} simply *terminates* with a success if it is provable.

In this paper, we make the above operational semantics more “constructive” and “interactive” by adopting the game semantics in [2, 3]. That is, our approach in this paper involves a modification of the operational semantics to allow for more active participation from the user. Solving $\forall xD$ from a program \mathcal{D} now has the following two-step operational semantics:

- Step (1): the machine tries to prove $\forall xD$ from a program \mathcal{D} . If it fails, the machine returns the failure. If it succeeds, goto Step (2).
- Step (2): the machine requests the user to choose a constant c for x and then proceeds with solving the goal, $[c/x]D$.

As an illustration of this approach, let us consider the following program.

$$\begin{aligned} &\{ emp(tom) \triangleq \top. emp(pete) \triangleq \top. \\ &boss(tom, bob) \triangleq \top. boss(pete, bob) \triangleq \top. \\ &wife(tom, mary) \triangleq \top. wife(pete, ann) \triangleq \top. wife(john, sue) \triangleq \top. \} \end{aligned}$$

As a particular example, consider a goal task $\forall x(emp(x) \supset \exists y wife(x, y))$. This goal simply terminates with a success in the context of [10] as it is solvable. However, in our context, execution requires more. To be specific, execution proceeds as follows: the system requests the user to select a particular employee for x . After the employee – say, *tom* – is selected, the system returns $y = mary$. As seen from the example above, universally quantified goals in intuitionistic logic can be used to model the *read* predicate in Prolog.

We also introduce *blind* universal quantifiers of the form $\forall^b xD$. This quantification is similar to $\forall xD$ but is read as “for an unknown value for x ”. The machine therefore does *not* request the user to choose any value for x for this quantification. As an illustration of this quantifier, let us consider a goal task $\exists y \forall^b x(emp(x) \supset boss(x, y))$. In this case, execution proceeds

as follows: the system chooses *bob* for y and then successfully terminates without requesting the user to choose a value for x .

In this paper we present the syntax and semantics of this language called $\text{Prolog}^{0/1}$. The remainder of this paper is structured as follows. We describe a subset of LINC logic in the next section. Section 3 describes the new semantics. Section 4 concludes the paper.

2 An Overview of Level 0/1 prover

Our language is a variant of a subset of the level 0/1 prover in [10], which is a simple fragment of LINC. Therefore, we closely follow their presentation in [10]. The language can also be seen as a version of Horn clauses with some extensions. We assume that a program – a set of definition clauses \mathcal{D} – is given. We have two kinds of goals given by G - and D -formulas below:

$$G ::= \top \mid \perp \mid A \mid G \wedge G \mid G \vee G \mid \exists x G$$

$$D ::= \top \mid \perp \mid A \mid D \wedge D \mid D \vee D \mid \exists x D \mid \forall x D \mid \forall^b x D \mid G \supset D$$

In the rules above, A represents an atomic formula.

The formulas in this languages are divided into *level-0* goals, given by G above, and *level-1* goals, given by D . We assume that atoms are partitioned level-0 atoms and level-1 atoms. Goal formulas can be level-0 or level-1 formulas, and in a definition $A \triangleq B$, A and B can be level-0 or level-1 formulas, provided that $\text{level}(A) \geq \text{level}(B)$.

Level-0 formulas and Level-1 formulas are similar to goal formulas in Prolog. However, when the Level-1 prover meets the implication $G \supset D$, it attempts to solve G . If G is solvable with all the possible answer substitutions $\sigma_1, \dots, \sigma_n$, then the Level-1 prover checks that, for every substitution σ_i , $D\sigma$ holds. If Level-0 finitely fails, the implication is proved.

We will present the standard operational semantics for this language as inference rules [1]. Below the notation $G : \mathcal{G}$ denotes $\{G\} \cup \mathcal{G}$. Note that execution alternates between two phases: the left rules phase and the right rules phase. In this fragment, all the left rules are invertible and therefore the left-rules take precedence over the right rules. Note that our semantics is a lazy version of the semantics of level 0/1 prover in the sense that an

answer substitution is applied as lazily as possible. This makes it easy to transit smoothly to the game-based execution model in the next section.

Definition 1. Let σ be an answer substitution and let G, D be a goal and let \mathcal{G} be a set of G -formulas. Then the task of proving D from an empty set with respect to $\sigma, \mathcal{D} - pv(\sigma, \emptyset \vdash D)$ (level 1)– and the task of proving D from \mathcal{G} with respect to $\sigma, \mathcal{D} - pv(\sigma, \mathcal{G} \vdash D)$ (level 0)– are (mutual recursively) defined as follows:

- (1) $pv(\sigma, \perp : \mathcal{G} \vdash D)$. % This is a success.
- (2) $pv(\sigma, \top : \mathcal{G} \vdash D)$ if $pv(\sigma, \mathcal{G} \vdash D)$. % \top in the premise is redundant.
- (3) $pv(\sigma, A : \mathcal{G} \vdash D)$ if, for each θ which is the $mgu(A\sigma, A')$ for some $A' \triangleq B \in \mathcal{D}$, $pv(\sigma\theta, B : \mathcal{G} \vdash D)$. % DefL rule
- (4) $pv(\sigma, G_0 \wedge G_1 : \mathcal{G} \vdash D)$ if $pv(\sigma, G_0 : \mathcal{G} \vdash D)$.
- (5) $pv(\sigma, G_0 \vee G_1 : \mathcal{G} \vdash D)$ if $pv(\sigma, G_0 : \mathcal{G} \vdash D)$ and $pv(\sigma, G_1 : \mathcal{G} \vdash D)$.
- (6) $pv(\sigma, \exists x G : \mathcal{G} \vdash D)$ if $pv(\sigma, [y/x]G : \mathcal{G} \vdash D)$ where y is a *new* free variable.
 % Below is the description of the level-1 prover
- (7) $pv(\sigma, \emptyset \vdash \top)$. % solving a true goal
- (8) $pv(\sigma, \emptyset \vdash A)$ if $A' \triangleq B \in \mathcal{D}$ and $A'\theta = A\sigma$ and $pv(\sigma\theta, \emptyset \vdash B)$. % DefR
- (9) $pv(\sigma, \emptyset \vdash D_0 \wedge D_1)$ if $pv(\sigma, \emptyset \vdash D_0)$ and $pv(\sigma, \emptyset \vdash D_1)$.
- (10) $pv(\sigma, \emptyset \vdash D_0 \vee D_1)$ if $pv(\sigma, \emptyset \vdash D_i)$ where i is 0 or 1.
- (11) $pv(\sigma, \emptyset \vdash G \supset D)$ if $pv(\sigma, G : \emptyset \vdash D)$. % switch from level 1 to level 0
- (12) $pv(\sigma, \emptyset \vdash \forall x D)$ if $pv(\sigma, \emptyset \vdash [y/x]D)$ where y is a *new* free variable.
- (13) $pv(\sigma, \emptyset \vdash \forall^b x D)$ if $pv(\sigma, \emptyset \vdash [y/x]D)$ where y is a *new* free variable.
- (14) $pv(\sigma, \emptyset \vdash \exists x D)$ if $pv(\sigma\sigma_1, \emptyset \vdash [w/x]D)$ where w is a new free variable, $\sigma_1 = \{\langle w, t \rangle\}$ and t is a term.

Most rules are straightforward to read.

The following is a proof tree of the example given in Section 1. Below the proof tree is represented as a list. Now, given σ, \mathcal{G} and D , a proof tree of a *proof formula* (σ, \mathcal{G}, D) is a list of tuples of the form $\langle E, Ch \rangle$ where E is a proof formula and Ch is a list of the form $i_1 :: \dots :: i_n :: nil$ where each i_k is the address of its k th child (actually the distance to E 's k th children in the proof tree).

```
{(h0, tom), (w0, ann)}, ∅ ⊢ ⊤, nil % success
{(h0, tom), (w0, ann)}, ∅ ⊢ wife(h0, w0), 1::nil % defR
{(h0, pete)}, ∅ ⊢ ∃y wife(h0, y), 1::nil % ∃-R
{(h0, tom), (w0, mary)}, ∅ ⊢ ⊤, nil % success
{(h0, tom), (w0, mary)}, ∅ ⊢ wife(h0, w0), 1::nil % defR
{(h0, tom)}, ∅ ⊢ ∃y wife(h0, y), 1::nil % ∃-R
∅, emp(h0) ⊢ ∃y wife(h0, y), 4::1::nil % defL
∅, ∅ ⊢ emp(h0) ⊃ ∃y wife(h0, y), 1::nil
∅, ∅ ⊢ ∀x(emp(x) ⊃ ∃y wife(x, y)), 1::nil % ∀-R
```

3 An Alternative Operational Semantics

Adding game semantics requires two execution phases: (1) the proof phase and (2) the execution phase. To be precise, our new execution model – adapted from [2] – actually solves the goal relative to the program using the proof tree built in the proof phase.

In the execution phase, to deal with the universally quantified goals properly, the machine needs to maintain an *input substitution* F of the form $\{y_0/c_0, \dots, y_n/c_n\}$ where each y_i is a variable introduced by a universally quantified goal in the proof phase and each c_i is a constant typed by the user during the execution phase.

Definition 2. Let i be an index, let L be a proof tree, let F be an input substitution. Then executing L_i (the i element in L) with F – written as $ex(i, L, F)$ – is defined as follows:

- (1) $ex(i, L, F)$ if $L_i = (E, nil)$. % no child, success.
- (2) $ex(i, L, F)$ if $L_i = (\sigma, \emptyset \vdash D_0 \wedge D_1, m :: 1 :: nil)$ and $ex(i - m, L, F)$ and $ex(i - 1, L, F)$. % two children

- (3) $ex(i, L, F)$ if $L_i = (\sigma, G_0 \vee G_1 : \mathcal{G} \vdash D, m :: 1 :: nil)$ and $ex(i - m, L, F)$ and $ex(i - 1, L, F)$. % two children
- (4) $ex(i, L, F)$ if $L_i = (\sigma, \emptyset \vdash \forall x D, 1 :: nil)$ and $L_{i-1} = (\sigma, \emptyset \vdash [y/x]D, Ch)$ and $read(k)$ and $ex(i - 1, L, F \cup \{y/c\})$ where c is the user input (the value stored in k). % update F for universal quantifiers.
- (5) $ex(i, L, F)$ if $L_i = (\sigma, \emptyset \vdash \exists x D, 1 :: nil)$ and $L_{i-1} = (\sigma, \emptyset \vdash [w/x]D, Ch)$ and $print(x = w\sigma F)$ and $ex(i - 1, L, F \cup \{y/c\})$
- (6) $ex(i, L, F)$ if $L_i = (\sigma, A : \mathcal{G} \vdash D, i_1 :: \dots :: i_n :: nil)$ and choose a i_k such that $L_{i-i_k} = (\sigma\theta_k, B : \mathcal{G} \vdash D, Ch)$ and (F and θ_k agree on the variables appearing in F) and $ex(i - i_k, L, F)$. % choose a correct one among many paths in defL
- (7) $ex(i, L, F)$ if $L_i = (E, 1 :: nil)$ and $ex(i - 1, L, F)$. % otherwise

Initially, F is an empty substitution. The following is an execution sequence of the goal $\forall x(emp(x) \supset \exists y wife(x, y))$ using the proof tree above. We assume here that the user chooses *pete* for x . Note that the last component represents F .

$\{(w_0, ann)\}, \top \vdash \top, \{(h_0, pete)\}$ % success
 $\{(w_0, ann)\}, \top \vdash wife(h_0, w_0), \{(h_0, pete)\}$ % defR
 $\emptyset, \top \vdash \exists y wife(h_0, y), \{(h_0, pete)\}$ % \exists -R
 $\emptyset, emp(h_0) \vdash \exists y wife(h_0, y), \{(h_0, pete)\}$ % defL
 $\emptyset, \emptyset \vdash emp(h_0) \supset \exists y wife(h_0, y), \{(h_0, pete)\}$ % update F
 $\emptyset, \emptyset \vdash \forall x(emp(x) \supset \exists y wife(x, y)), \emptyset$ % \forall -R

4 Conclusion

In this paper, we have considered a new execution model for the level 0/1 prover. This new model is interesting in that it gives a logical status to the *read* predicate in Prolog. We plan to connect our execution model to Japaridze's Computability Logic [2, 3] in the near future.

References

- [1] G. Kahn, “Natural Semantics”, In the 4th Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 247, 1987.
- [2] G. Japaridze, “Introduction to computability logic”, Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.
- [3] G. Japaridze, “Sequential operators in computability logic”, Information and Computation, vol.206, No.12, pp.1443–1475, 2008.
- [4] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.
- [5] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. *A User Guide to Bedwyr*, November 2006.
- [6] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Proc. LICS 1997*, pp. 434–445, IEEE Comp. Soc. Press, 1997.
- [7] Peter Schroeder-Heister. Rules of definitional reflection. In *Proc. LICS 1993*, pages 222–232. IEEE Comp. Soc. Press, 1993.
- [8] Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
- [9] Alwen Tiu. Model checking for π -calculus using proof search. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
- [10] Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Proc. of ESHOL’05: Empirically Successful Automated Reasoning in Higher-Order Logics*, pages 79 – 98, December 2005.